J Autom Reasoning (2015) 55:91–116
DOI 10.1007/s10817-015-9329-1

MaLeS: A Framework for Automatic Tuning of Automated Theorem Provers

Daniel Kühlwein[1] · **Josef Urban**[1]

Received: 13 August 2013 / Accepted: 5 May 2015 / Published online: 16 July 2015

Abstract MaLeS is an automatic tuning framework for automated theorem provers. It provides solutions for both the strategy finding as well as the strategy scheduling problem. This paper describes the tool and the methods used in it, and evaluates its performance on three automated theorem provers: E, LEO-II and Satallax. On a representative subset of the TPTP library a MaLeS-tuned prover solves on average 8.67 % more problems than the prover with its default settings.

Keywords Strategy selection · Machine learning · Automated theorem provers

1 Introduction

Automated theorem proving is a search problem. Many different approaches exist, and most of them have parameters that can be tuned. Examples of such parameters are clause weighting and selection schemes, term orderings, and sets of inference and reduction rules used. For a given automated theorem prover (ATP) A, its parameters form A's *parameter space*. A specific choice of parameters is called a *strategy*,[1] i.e., strategies are elements of the

[1] Unfortunately, there is no standard terminology for this. In Satallax [4] parameters are called flags, and a strategy is called a mode. Option can be used as synonym for parameter. Configurations and configuration space are other alternative names.

✉ Daniel Kühlwein
daniel.kuehlwein@gmail.com

Josef Urban
josef.urban@gmail.com

[1] Intelligent Systems, Institute for Computing and Information Sciences, Radboud University Nijmegen, Nijmegen, Netherlands

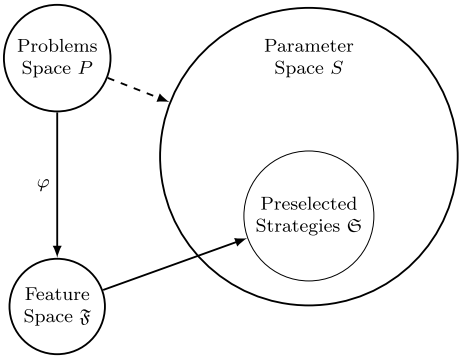

Fig. 1 Overview of the strategy selection problem for ATPs

parameter space (Fig. 1). The choice of strategy can often make the difference between finding a proof within a few milliseconds or not at all (within a reasonable time limit). This naturally raises to the question: Given a new problem, which search strategy should be used?

This problem has already received a considerable amount of attention. Gandalf [25] pioneered *strategy scheduling*: Instead of running a single strategy for the entire time span specified by the user, several search strategies are run sequentially for shorter times. This method is used in most current ATPs, most prominently Vampire [13]. In the SETHEO project [28], a local search algorithm was used to find better sequences of strategies. Fuchs [5] employed a nearest neighbor algorithm to determine which strategy/ies to run. Bridge's [3] thesis is concerned with machine learning for search heuristic selection in ATPs with a particular focus on problem features and feature selection. In the SAT community, Satzilla [29] very successfully used machine learning to decide when to run which SAT solver. ParamILS [7] is a general tuning framework that searches for good parameter settings with a randomized hill climbing algorithm. BliStr [26] uses ParamILS to develop strategies for E [15] on a large set of interrelated problems.

Despite all this work, most ATPs do not harness the methods available. Search strategies are often manually defined by the developer of the ATP and strategy schedules are created by a greedy algorithm or very simple clustering. This paper introduces *MaLeS* (Machine Learning (of) Strategies), a learning-based framework for automatic tuning and configuration of ATPs. It is based on and supersedes E-MaLeS 1.0 [10] and E-MaLeS 1.1 [8]. The goal of MaLeS is to help ATP users fine-tune an ATP to their problems, and give developers a simple tool for finding good search strategies and creating strategy schedules. MaLeS is implemented in Python and has been tested with the ATPs E, LEO-II [1] and Satallax [4]. The source code is freely available at https://github.com/dkuehlwein/males.

1.1 The Strategy Selection Problem

Figure 1 gives an informal overview of the *strategy selection problem*. Given an ATP problem $p \in P$, find a *strategy* s in the *parameter space* S that can quickly solve p. First, we note that parameter spaces can be very big. For example, the ATP E supports over 10^{17} different

strategies. Hence, to simplify the strategy selection problem, strategy selection algorithms usually consider only a small number of *preselected strategies* $\mathfrak{S}$. Defining $\mathfrak{S}$ is the first challenge. There are different criteria to determine which strategies should be selected. The most common one is to pick strategies that solve a lot of problems, or are very good for a particular kind of problem.

As a second step, we need a way to characterize problems. This is usually accomplished by defining a set of *features* $\mathfrak{F}$. The features must strike a balance between being fast to compute (via a *feature function* φ) and being expressive enough so that the ATP behaves similarly on problems with similar features. Once the features are defined, we need a way to predict how well each preselected strategy performs on ATP problems with particular features. Finally, the predictions need to be combined into a strategy schedule. Hence, the strategy selection problem consists of three subproblems:

- Finding a *good* set of preselected strategies $\mathfrak{S}$.
- Defining features $\mathfrak{F}$ which are easy to compute, but also expressive enough to distinguish different types of problems.
- Determining a method which given the features of a problem creates a strategy schedule.

1.2 Overview

The rest of the paper is organized as follows: Section 2 explains how MaLeS defines the preselected strategies $\mathfrak{S}$. The features and the algorithm that creates the strategy schedule are presented in Section 3. MaLeS is evaluated against E 1.7, LEO-II 1.6.0 and Satallax 2.7 run with default settings in Section 4. The experiments compare the performance of running an ATP in default mode versus running the ATP with strategy scheduling provided by MaLeS. Future work is considered in Section 5, and the paper concludes with Section 6. The Appendix shows how to install the MaLeS-tuned versions of the ATPs mentioned above: E-MaLeS, LEO-MaLeS and Satallax-MaLeS, how to tune any of those systems for new problems, and how to use MaLeS with different ATPs. It also includes an overview of the CADE ATP System Competition (CASC) [24] results.

2 Finding Good Search Strategies with MaLeS

Choosing a good strategy for a problem requires prior information on how the different strategies behave on different kinds of problems. Acquiring this information for all strategies is often infeasible due to constraints on CPU power available and the number of possible strategies. Hence, we have to decide which strategies we want to evaluate. ATP developers often manually define such a set of strategies based on their intuition and experience. This option is, however, not available when one lacks in-depth knowledge of the internal workings of the ATP. A local search algorithm can help in these cases, and can even be combined with the manual approach by taking the predefined strategies as starting points of the search.

Algorithm 1 find_strategies: For each problem search for the strategy that solves it in the least amount of time.

```
 1: procedure FIND_STRATEGIES(Problems,tol,t_max,nS,nC)
 2:     initialize Queue Q // See description
 3:     initialize dictionary bestTime with t_max for all problems
 4:     initialize dictionary bestStrategy as empty
 5:     while Q not empty do
 6:         s ← pop(Q)
 7:         for p ∈ Problems do
 8:             oldBestTime ← bestTime[p]
 9:             proofFound,timeNeeded ← run_strategy(s, p,t_max)
10:             if proofFound and timeNeeded < bestTime[p] then
11:                 bestTime[p] ← timeNeeded
12:                 bestStrategy[p] ← s
13:             end if
14:             if proofFound and timeNeeded < bestTime[p]+tol then
15:                 randomStrategies ← create_random_strategies(s,nS,nC)
16:                 for r in randomStrategies do
17:                     proofFoundR,timeNeededR ← run_strategy(r, p, timeNeeded)
18:                     if proofFoundR and timeNeededR<bestTime[p] then
19:                         bestTime[p] ← timeNeededR
20:                         bestStrategy[p] ← r
21:                     end if
22:                 end for
23:                 if bestTime[p] < oldBestTime then
24:                     Q ← put(Q,bestStrategy[p])
25:                 end if
26:             end if
27:         end for
28:     end while
29:     return bestStrategy
30: end procedure
```

The initialization of Q in Line 2 is done either by randomly creating some strategies, or by manually defining which strategies to use. Variable `tol` defines the tolerance of the algorithm, `t_max` is the maximal time that may be used by the strategy. `nS` determines the number of strategies generated in the `create_random_strategies` sub-procedure, `nC` is an upper limit for the number of different parameters between the new strategies and the original strategy. `bestStrategy` is a dictionary that for each problems stores the strategy that solved it in the least amount of time.

MaLeS employs a basic stochastic local search algorithm which we call *find_strategies* (Algorithm 1). The strategies returned by *find_strategies* define the *preselected strategies* $\mathfrak{S}$. The difference between *find_strategies* and existing parameter selection frameworks like ParamILS and BliStr is that *find_strategies* searches for the fastest strategy for each problem, whereas ParamILS tries to find the best strategy for all problems (i.e. find the strategy that solves the highest number of problems within some time limit).[2] BliStr searches for the best strategy for sets of similar problems.

find_strategies takes a list of problems as input. A queue of start strategies is initialized, either with random or predefined strategies. Each strategy in the queue is then used on all problems. If a strategy solves a problem faster than any of the previously tried strategies (within some tolerance range, see Line 14), a local search is performed. If the search yields

[2] *find_strategies* is essentially equivalent to running ParamILS on every single problem.

faster strategies, the fastest of all newly found search strategies is appended to the queue. In the end, *find_strategies* returns all strategies that were fastest on at least one problem.

Algorithm 2 create_random_strategies: Returns slight variations of the input strategy.

```
1: procedure CREATE_RANDOM_STRATEGIES(Strategy,nS,nC)
2:     newStrategies is an empty list
3:     _i = 1
4:     while _i ≤ nS do
5:         newStrategy is a copy of Strategy
6:         _j = 1
7:         while _j ≤ nC do
8:             newStrategy = change_random_parameter(newStrategy)
9:             _j = _j + 1
10:        end while
11:        newStrategies.append(newStrategy)
12:        _i = _i + 1
13:    end while
14:    return newStrategies
15: end procedure
```

`nS` determines the number of new strategies, `nC` is the upper limit for the number of changed parameters.

The local search part is defined in Algorithm 2 (*create_random_strategies*). It returns a predefined number `nS` of strategies similar to the input strategy. The new strategies are created by randomly changing the parameters of the input strategy. How many parameters are changed is determined by `nC`.

3 Strategy Scheduling with MaLeS

Many current ATPs use some kind of strategy scheduling as a default mode of operation. Some use the same schedule for every problem (e.g. Satallax 2.7). Others define classes of similar problems and use different schedules for different classes (e.g. E 1.7, LEO-II 1.6.0). MaLeS creates an individual strategy schedule for each problem. This is done by learning which strategies are useful for which type of problems.

3.1 Notation

We use the following notation:

- p is an ATP problem. P denotes a set of problems.
- $P_{\text{train}} \subseteq P$ is a set of training problems that is used to tune the learning algorithm.
- $\mathfrak{F}$ is the feature space. We assume that $\mathfrak{F}$ is a subset of $\mathbb{R}^n$ for some $n \in \mathbb{N}$.
- $\varphi : P \to \mathfrak{F}$ is the *feature function*. $\varphi(p)$ is the feature vector of a problem.
- S is the parameter space, $\mathfrak{S}$ is the set of preselected strategies.
- The time the ATP running strategy s needs to solve a problem p is denoted by $\tau(p, s)$. If s is obvious from the context or irrelevant, we also use $\tau(p)$.
- For a strategy s, $\rho_s : P \to \mathbb{R}$ its runtime *prediction function*. For each strategy s in the preselected strategies $\mathfrak{S}$, MaLeS defines a runtime prediction function $\rho_s : P \to \mathbb{R}$. The prediction function ρ_s uses the features of a problem to predict the time the ATP

running strategy s needs to solve the problem. The strategy schedule for the problem is created from these predictions.

3.2 Features

Features provide an abstract description of a problem. Optimally, the features should be designed in such a way that the ATP behaves similarly on problems with similar features, i.e. if two problem p, q have similar features $\varphi(p) \sim \varphi(q)$, then for each strategy s the runtimes should be similar $\tau(p, s) \sim \tau(q, s)$. The similarity function (e.g. cosine distance between the feature vectors) and set of features heavily influence the quality of the prediction functions. Indeed, feature selection is an entire subfield of machine learning [6, 11].

Currently, MaLeS supports two different feature spaces: Schulz's E features are used for first order (FOF) problems. The TPTP features designed by Sutcliffe are used for higher order (THF) problems [22]. Note that the main reason for using these features was that they were easily available. Evaluating different feature sets and/or introducing new features is beyond the scope of this paper.

3.2.1 The E Features

Schulz designed a set of features for clause-normal-form and first order problems. They are used in the strategy selection process in his theorem prover E [15]. Table 1 shows the features together with a short description of each.[3] MaLeS uses the same features for first-order problems.

The features are computed by running Schulz's *classify_problem* program which is distributed with MaLeS.

3.2.2 The TPTP Features

For every problem, the TPTP problem library [17] provides a syntactical description which can be used as problem features. Figure 2 shows an example. Before normalization, the feature vector corresponding to the example is

$$[145, 5, 47, 31, 1106, \ldots, 147, 0, 0, 0, 0]$$

Sutcliffe's *MakeListStats* computes these features and is publicly available as part of the TPTP infrastructure. A modified version that outputs only the numbers without any text is also distributed with MaLeS.

3.2.3 Normalization

In the initial form, there can be great differences between the values of different features. In the THF example (Fig. 2), the number of atoms (1106) is of a different order of magnitude than e.g. the maximal formula depth (7). Since our machine learning method (like many other) computes the Euclidean distance between data points, these differences can render smaller valued features irrelevant. Hence, normalization is used to scale all features to have values between 0 and 1. First the features for each $p \in P_{\text{train}}$ are computed. Then the

[3]The authors would like to thank Stephan Schulz for the design of the features, the program that extracts them, and their description in this subsection.

Table 1 Problem features used for strategy selection in E and in first-order MaLeS

Feature	Description
`axioms`	Most specific class (unit, Horn, general) describing all axioms
`goals`	Most specific class (unit, Horn) describing all goals
`equality`	Problem has no equational literals, some equational literals, or only equational literals
`non-ground units`	Number (or fraction) of unit axioms that are not ground
`ground-goals`	Are all goals ground?
`clauses`	Number of clauses
`literals`	Number of literals
`term_cells`	Number of all (sub)terms
`unitgoals`	Number of unit goals (negative clauses)
`unitaxioms`	Number of positive unit clauses
`horngoals`	Number of Horn goals (non-unit)
`hornaxioms`	Number of Horn axioms (non-unit)
`eq_clauses`	Number of unit equations
`groundunitaxioms`	Number of ground unit axioms
`groundgoals`	Number of ground goals
`groundpositiveaxioms`	Number (or fraction) of positive axioms that are ground
`positiveaxioms`	Number of all positive axioms
`ng_unit_axioms_part`	Number of non-ground unit axioms
`max_fun_arity`	Maximal arity of a function or predicate symbol
`avg_fun_arity`	Average arity of symbols in the problem
`sum_fun_arity`	Sum of arities of symbols in the problem
`clause_max_depth`	Maximal clause depth
`clause_avg_depth`	Average clause depth

maximal and minimal value of each feature f is determined. These values are then used to rescale the feature vectors for each problem p via

$$\varphi(p)_f := \frac{\varphi(p)_f - \min_f}{\max_f - \min_f}$$

where $\varphi(p)_f$ is the value of feature f for problem p, $\min_f$ is the minimal and $\max_f$ is the maximal value for f among the problems in P_{train}.

```
% Syntax   : Number of formulae    :  145 (   5 unit;  47 type;  31 defn)
%            Number of atoms       : 1106 (  36 equality; 255 variable)
%            Maximal formula depth :   11 (   7 average)
%            Number of connectives :  760 (   4   ~;   4   |;   8   &; 736   @)
%                                         (   0 <=>;   8  =>;   0  <=;   0 <~>)
%                                         (   0  ~|;   0  ~&;   0  !!;   0  ??)
%            Number of type conns  :  235 ( 235   >;   0   *;   0   +;   0  <<)
%            Number of symbols     :   52 (  47   :)
%            Number of variables   :  147 (   3 sgn;  29   !;   6   ?; 112   ^)
%                                         ( 147   :;   0  !>;   0  ?*)
%                                         (   0  @-;   0  @+)
```

Fig. 2 The TPTP features of the THF problem AGT029^1.p in TPTP-v5.4.0

3.3 Runtime Prediction Functions

Predicting the runtime of an ATP is a classic regression problem [2]. For each strategy s in the preselected strategies $\mathfrak{S}$, we are searching for a function $\rho_s : P \to \mathbb{R}$ such that for all problems $p \in P$ the predicted values are close to the actual runtimes: $\rho_s(p) \sim \tau(p, s)$. This section explains the learning method employed by MaLeS as well as the data preparation techniques used.

3.3.1 Timeouts

The prediction functions are learned from the behavior of the preselected strategies on the training problems P_{train}. Each preselected strategy is run on all training problems with a timeout t. Often, strategies will not solve all problems within the timeout. This leads to the question how one should treat unsolved problems. Setting the time value of an unsolved problem-strategy pair (p, s) to the timeout, i.e. $\tau(p, s) = t$, is one possible solution. Another possibility, which is used in MaLeS, is to learn on only problems that can be solved. While ignoring unsolved problems introduces a bias towards shorter runtimes, it also simplifies the computation of the prediction functions and allows to update the prediction functions at runtime (Section 3.5).

3.3.2 Kernel Methods

MaLeS uses *kernels* to learn the runtime prediction function. Kernels are a popular machine learning method and have successfully been applied in many domains [16]. A kernel can be seen as a similarity function between feature vectors. Kernels allow the use of nonlinear features while keeping the learning problem itself linear. The basic principles will be covered below. More information about kernel-based machine learning can be found in [16].

Definition 1 (Gaussian Kernel) The Gaussian kernel k with parameter σ of two problems $p, q \in P$ with feature vectors $\varphi(p), \varphi(q) \in \mathfrak{F} \subseteq \mathbb{R}^n$ for some $n \in \mathbb{N}$ is defined as

$$k(p, q) := \exp\left(-\frac{\varphi(p)^T\varphi(p) - 2\varphi(p)^T\varphi(q) + \varphi(q)^T\varphi(q)}{\sigma^2}\right)$$

$\varphi(p)^T$ is the transposed vector, and hence $\varphi(p)^T\varphi(q)$ is the dot product between $\varphi(p)$ and $\varphi(q)$ in $\mathbb{R}^n$.

In order to apply machine learning, first some data to learn from is required. Let $t \in \mathbb{R}$ be a time limit. For each preselected strategy $s \in \mathfrak{S}$, the ATP is run with strategy s and time limit t on each problem in P_{train}. Note that the same t is used for all problems. For each strategy s, $P^s_{\text{train}} \subseteq P_{\text{train}}$ is the set of problems that the ATP can solve within the time limit t with strategy s.

Definition 2 (The Prediction Function) In kernel based machine learning, the prediction function ρ_s has the form

$$\rho_s(p) = \sum_{q \in P^s_{\text{train}}} \alpha^s_q k(p, q)$$

for some $\alpha^s_q \in \mathbb{R}$. The α^s_q are called weights and are the result of the learning. To define how exactly this is done, some more notation is needed.

Definition 3 (Kernel Matrix, Times Matrix and Weights Matrix) For every strategy $s \in \mathfrak{S}$, let m be the number of problems in P^s_{train} and $(p_i)_{i \in m}$ be an enumeration of the problems in P^s_{train}. The kernel matrix $K^s \in \mathbb{R}^{m \times m}$ is defined as

$$K^s_{i,j} := k(p_i, p_j)$$

We define the time matrix $Y^s \in \mathbb{R}^{1 \times m}$ via

$$Y^s_i := \tau(p_i, s)$$

Finally, we set the weight matrix $A^s \in \mathbb{R}^{m \times 1}$ as

$$A^s_i := \alpha^s_{p_i}$$

If it is obvious which strategy is referred to, or the statement is independent of the strategy, we omit the ^{s} in K^s,Y^s and A^s.

A simple way to define values for the weights $\alpha^s_{p_i}$ would be to solve $KA = Y$. Such a solution (if it exists) would likely perform very well on known data but poorly on new data, a behavior called *overfitting*. As a measure against overfitting, a regularization parameter $\lambda \in \mathbb{R}$ is added and least square regression is used to minimize the difference between the predicted times and the actual times [14]. That means we want

$$A = \underset{A \in \mathbb{R}^{m \times 1}}{\arg\min} \left((Y - KA)^T (Y - KA) + \lambda A^T K A \right)$$

The first part of the equation $(Y - KA)^T (Y - KA)$ is the square loss between the predicted values and the actual time needed. $\lambda A^T K A$ is the regularization term. $A^T K A$ is a measure of how complex, in terms of VC dimension [27], our prediction function is. The bigger λ, the more complex functions are penalized. For very high values of λ, we force A to be almost equal to the 0 matrix. This approach can be seen as a kind of Occam's razor for prediction functions. A is the matrix that best fits the training data while remaining as simple as possible.

Theorem 1 (Weight Matrix for a Strategy) *For $\lambda > 0$, the optimal weights for a strategy s are given by*

$$A = (K + \lambda I)^{-1} Y$$

with I being the identity matrix in $\mathbb{R}^{m \times m}$.

Proof

$$\begin{aligned} & \tfrac{\partial}{\partial A} \left((Y - KA)^{\mathrm{T}} (Y - KA) + \lambda A^{\mathrm{T}} K A \right) \\ = \quad & -2K(Y - KA) + 2\lambda KA \\ = \quad & -2KY + (2KK + 2\lambda K)A \end{aligned}$$

It can be shown that K is a positive-semi definite symmetric matrix and therefore $(K + \lambda I)$ is invertible for $\lambda > 0$. To find a minimum, we set the derivative to zero and solve with respect to A.

$$K(K + \lambda I)A = KY$$

and hence

$$A = (K + \lambda I)^{-1} Y$$

is a solution. □

3.4 Crossvalidation

Finally, the values for the regularization constant λ and the kernel width σ need to be determined. This is done via 10-fold *cross-validation* on the training problems, a standard machine learning method for such tasks [9]. Cross-validation simulates the effect of not knowing the data and picks the values that perform, in general, best on unknown problems.

First a finite number of possible values for λ and σ is defined. Then, the training set P^s_{train} is split into 10 disjoint, equally sized subsets $P_1, \ldots P_{10}$. For all $1 \leq i \leq 10$, each possible combination of values for λ and σ is trained on $P^s_{\text{train}} - P_i$ and evaluated on P_i. The evaluation is done by computing the square-loss between the predicted runtimes and the actual runtimes. The combination with the least average square loss is used.

3.5 Creating Schedules from Prediction Functions

MaLeS uses the knowledge of how different strategies perform on a set of training problems to estimate how these strategies will behave on a new problem. This is done by learning runtime prediction functions as described above using the data gathered with Algorithm 1. With the runtime prediction functions we can create individual strategy schedules for new problems, i.e. compute a strategy schedule for every set of features.

Given a new problem, MaLeS iterates between computing the predicted runtimes for each strategy, running the predicted best strategy and updating the prediction models. Algorithm 3 shows the details.

Algorithm 3 males: Tries to solve the input problem within the time limit. Creates and runs a strategy schedule for the problem.

```
 1: procedure MALES(problem,time)
 2:     proofFound,timeUsed ← run_start_strategies(problem,time)
 3:     if proofFound then
 4:         return timeUsed
 5:     end if
 6:     while timeUsed < time do
 7:         Set times as an empty list
 8:         for s ∈ 𝔖 do
 9:             t_s ← ρ_s(problem)
10:             times.append([t_s, s])
11:         end for
12:         ([t_s', s']) ← choose_best_strategy(times)
13:         proofFound,timeNeeded ← run_strategy(s',problem, t_s')
14:         timeUsed = timeUsed + timeNeeded
15:         if proofFound then
16:             return timeUsed
17:         end if
18:         for s ∈ 𝔖 do
19:             timeUsed = timeUsed + update_prediction_function(ρ_s ,s' ,t_s')
20:         end for
21:     end while
22:     return timeUsed
23: end procedure
```

In line 2 the algorithm starts by running some predefined start strategies. The goal of running these start strategies first is to filter out simple problems, which allows the learning algorithm to focus on the more difficult problems. The start strategies are picked greedily. First the strategy that solves the highest number of problems (within some time limit) is chosen. Then the strategy that solves the highest number of problems that were not solved

by the strategy picked first (within some time limit) is selected, etc. The number of start strategies and their runtime are determined via their respective parameters in the *setup.ini* file (Table 2). Training problems that are solved by the start strategies are deleted from the training set. For example, let $s_1, \ldots, s_n$ be the starting strategies, all with a runtime of 1 second. Then for all $s \in S'$ we can set

$$P^s_{\text{train}} := \{p \in P^s_{\text{train}} \mid \forall\, 1 \leq i \leq n\ \tau(p, s_i) > 1\}$$

and train ρ_s on the updated P^s_{train}.

The subprocedure `choose_best_strategy` in line 12 picks the strategy with the minimum predicted runtime among those that have not been run with a bigger or equal runtime before.[4] `run_strategy` runs the ATP with strategy s' and time limit $t_{s'}$ on the `problem`. If the ATP cannot solve the problem within the time limit, this information is used to improve the prediction functions in `update_prediction_function` (Line 19). For this, all the training problems that are solved by the picked strategy s' within the predicted runtime $t_{s'}$ are deleted from the training set P_{train}, i.e. for all $s \in S'$

$$P^s_{\text{train}} := \{p \in P^s_{\text{train}} \mid \tau(p, s') > t_{s'}\}$$

Afterwards, new prediction functions are learned on the reduced training set. This is done by first creating a new kernel and time matrix for the new P^s_{train} and then computing new weights as shown in Theorem 1. Due to the small size of the training dataset, this can be done in real time during a proof. Note that these updates are local, i.e., do not have any effect on future calls to MALES. If MALES finds a proof, the total time needed is returned to the user.

4 Evaluation

MaLeS has been evaluated with three different ATPs: E 1.7, LEO-II 1.6 and Satallax 2.7. For each prover, a set of training and testing problems is defined. MaLeS first searches for good strategies on the training problems using Algorithm 1 with a 10 second time limit, i.e. $t_{\max} = 10$. Promising strategies are then run for 300 seconds on all training problems. The resulting data is used to learn runtime prediction functions and strategy schedules as explained in the previous section. After the learning, MaLeS uses Algorithm 3 when trying to solve a new problem. The difference between the different MaLeS versions (i.e. E-MaLeS, Satallax-MaLeS and Leo-MaLeS) is the training data used to create the prediction functions and start strategies, and the ATP that is run in the `run_strategy` part of Algorithm 3. The MaLeS version of the ATP is compared with the default mode on both the test and the training problems.

4.1 E-MaLeS

E is a popular ATP for first order logic. It is open source, easily available and consistently performs very well in CASC. Additionally, E is easily tunable with a big parameter space which suggests that parameter tuning could lead to significant improvements. All computations were carried out on a 64 core AMD Opteron Processor 6276 with 1.4GHz per CPU and 256 GB of RAM

[4] If there are several strategies with the same minimal predicted runtime a random one is chosen.

4.1.1 E's Automatic Mode

E's automatic mode is developed by Stephan Schulz and based on a static partitioning of the set of all problems into disjoint classes. It is generated in two steps. First, the set of all training examples (typically the set of all current TPTP problems) is classified into disjoint classes using some of the features listed in Table 1. For the numeric features, threshold values have originally been selected to split the TPTP into 3 or 4 approximately equal subsets on each feature. Over time, these have been manually adapted using trial and error.

Once the classification is fixed, a Python program assigns to each class one of the strategies that solves the highest number of examples in this class. For *large* classes (arbitrarily defined as having more than 200 problems), it picks the strategy that additionally is the fastest for this class on average. For small classes, it picks the globally best strategy among those that solve the maximum number of problems. A class with zero solutions by all strategies is assigned the overall best strategy.

4.1.2 The Training Data

The problems from the FOF divisions of CASC-22 [18], CASC-J5 [19], CASC-23 [20] and CASC-J6 and CASC@Turing [21] were used as training problems. Several problems appeared in more than one CASC. There are also a few problems from earlier CASCs that are not part of the TPTP version used in the experiments, TPTP-v5.4.0. Deleting duplicates and missing problems leaves 1112 problems that were used to train E-MaLeS. The strategy search for the set of preselected strategies took three weeks on a 64 core server. The majority of the time was spent running promising strategies with a 300 second time limit. Over 2 million strategies were considered. Of those, 109 were selected to be used in E-MaLeS. E-MaLeS runs 10 start strategies, each with a 1 second time limit. E 1.7 (running the automatic mode) and E-MaLeS were evaluated on all training problems with a 300 second time limit. The results can be seen in Fig. 3.

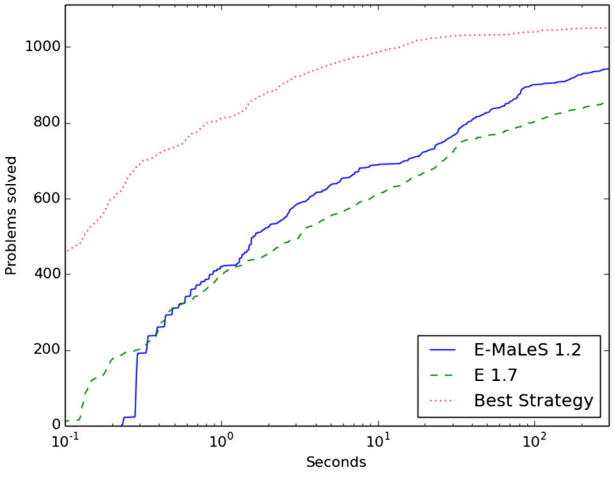

Fig. 3 Performance graph for E-MaLeS 1.2 on the training problems

Altogether, 1055, or 94.9 %, of the problems can be solved by E 1.7 with the strategies considered. E 1.7's automatic mode solves 856 of the problems (77.0 %), E-MaLeS solves 10.0 % more problems: 942 (84.7 %). *Best Strategy* shows the best possible result, i.e. the number of problems solved if for each problem the strategy that solves it in the least amount of time is selected.

4.1.3 The Test Data

Similar to the way the problems for CASC are chosen, 1000 random FOF problems of TPTP-v5.4.0 with a difficulty rating [23] between 0.2 and (including) 1.0 were chosen for the test dataset. 165 of the test problems are also part of the training dataset.

The results are similar to the results on the training problems and can be seen in Fig. 4. In the first three seconds, E solves more problems than E-MaLeS. Afterwards, E-MaLeS overtakes E. After 300 seconds, E-MaLeS solves 573 of the problems (57.3 %) and E 1.7 511 (51.1 %), an increase of 12.4 %. Figure 5 shows results for those 835 problems that are not part of the training set. The graphs are very similar which indicates that E-MaLeS did not overfit on the training data.

4.2 Satallax-MaLeS

In order to show that MaLeS also works for other ATPs, we picked a very different ATP for the next experiment: Satallax. Satallax is a higher order theorem prover that has a reputation of being highly tuned. The built-in strategy schedule of Satallax solves 95.3 % of all solvable problems in a training dataset (defined in Section 4.2.2) and, with the right parameters, 91.3 % (525) of the training problems can be solved in less than 1 second. The strategy search for the set of preselected strategies was carried out on a 32 core Intel Xeon with 2.6GHz per CPU and 256 GB of RAM. The evaluations were performed on a 64 core AMD Opteron Processor 6276 with 1.4GHz per CPU and 256 GB of RAM.

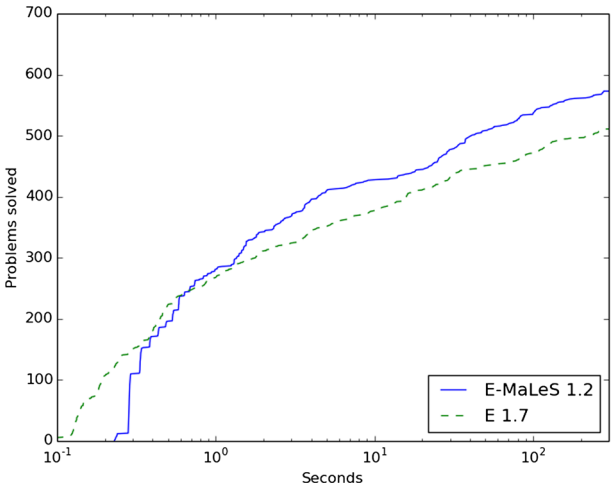

Fig. 4 Performance graph for E-MaLeS 1.2 on the test problems

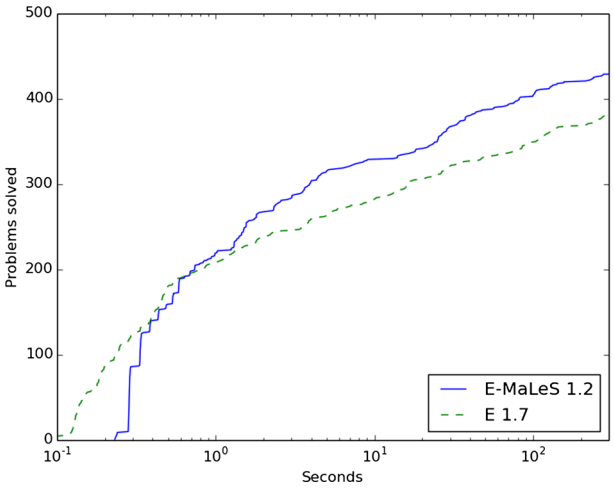

Fig. 5 Performance graph for E-MaLeS 1.2 on the unseen test problems

4.2.1 Satallax's Automatic Mode

Satallax employs a hard-coded strategy schedule that defines a sequence of strategies together with their runtimes. The same schedule is used for all problems. It is defined in the file *satallaxmain.ml* in the *src* directory of the Satallax installation. Many modes are only run for a very short time (0.2 seconds). This can cause problems if Satallax is run on CPUs that are slower than the one(s) used to create the schedule.

4.2.2 The Training Data

The problems from the THF divisions of CASC-J5 [19], CASC-23 [20] and CASC-J6 [21] were used as training problems. The THF division at CASC-J5 contained 200 problems, at CASC-23 300 problem, and at CASC-J6 another 200 problems. After deleting duplicates and problems that are not available in TPTP-v5.4.0, 573 problems remain. The strategy search took approximately 3 weeks. In the end, 111 strategies were selected to be used in Satallax-MaLeS. Satallax-MaLeS runs 20 start strategies, each with a 0.5 second time limit.

533 of the 573 problems are solvable with the appropriate strategy. Satallax and Satallax-MaLeS were evaluated on all training problems with a 300 second time limit. Satallax solves 508 of the problems (88.7 %). Satallax-MaLeS solves 1.6 % more problems for a total of 516 solved problems (90.1 %).

Figure 6 shows a log-scaled time plot of the results. For low time limits, Satallax-MaLeS solves significantly more problems than Satallax. It seems that Satallax's automatic mode is very suboptimal which might be a result of focusing on only the number of problems solved after 300 seconds. *Best Strategy* shows the best possible result, i.e. the number of problems solved if for each problem the strategy that solves it in the least amount of time is selected.

4.2.3 The Test Data

Similar to the E-MaLeS evaluation, the test dataset consists of 1000 randomly selected THF problems of TPTP-v5.4.0 with a difficulty rating between 0.2 and (including) 1.0.

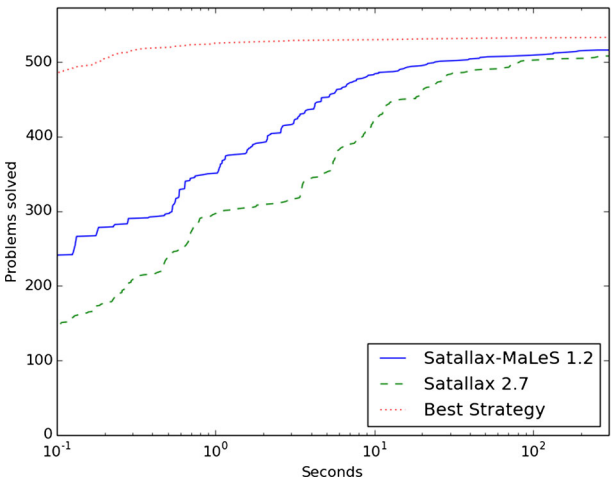

Fig. 6 Performance graph for Satallax-MaLeS 1.2 on the training problems

301 of the test problems are also part of the training dataset. The results are similar to the results on the training problems and can be seen in Fig. 7. While the end results are almost the same with Satallax-MaLeS solving 590 (59.0 %) and Satallax solving 587 (58.7 %) of the problems, Satallax-MaLeS significantly outperforms Satallax for lower time limits.

Figure 8 shows the results for those 699 problems that are not contained in the training problems. Here, Satallax-MaLeS solves more problems than Satallax in the beginning, but fewer for longer time limits. After 300 seconds, Satallax solves 344 and Satallax-MaLeS 336 problems.

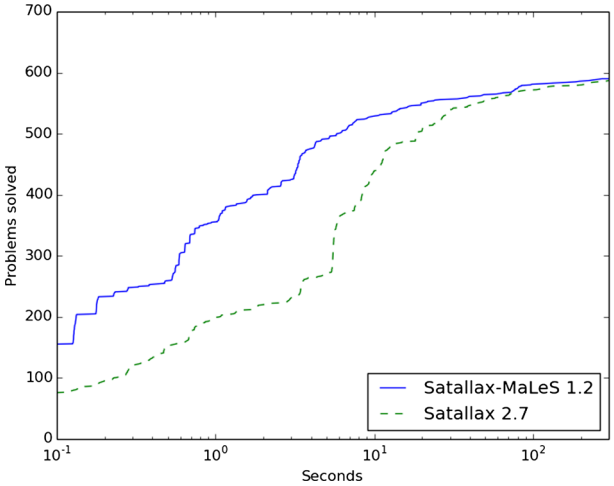

Fig. 7 Performance graph for Satallax-MaLeS 1.2 on the test problems

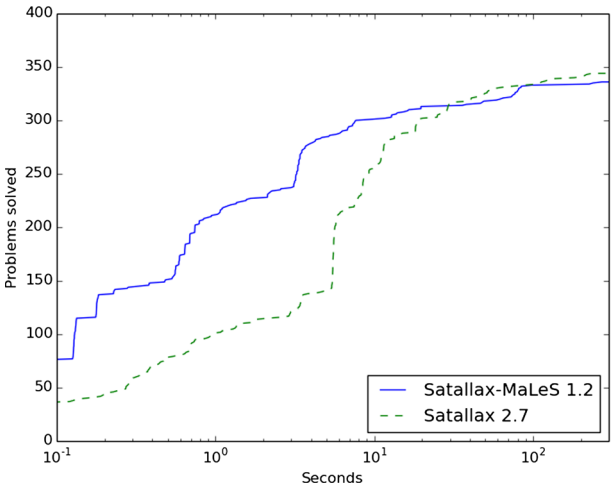

Fig. 8 Performance graph for Satallax-MaLeS 1.2 on the unseen test problems

4.3 LEO-MaLeS

LEO-MaLeS is the latest addition to the MaLeS family. LEO-II is a resolution-based higher-order theorem prover designed for fruitful cooperation with specialist provers for natural fragments of higher-order logic.[5] The strategy search for the set of preselected strategies and all evaluations were done on a 32 core Intel Xeon with 2.6GHz per CPU and 256 GB of RAM.

4.3.1 LEO-II's Automatic Mode

LEO-II's automatic mode is a combination of E's and Satallax's automatic modes. The problem space is split into disjoint subspaces and a different strategy schedule is used for each subspace. The automatic mode is defined in the file *strategy_scheduling.ml* in the *src/interfaces* directory of the LEO-II installation.

4.3.2 The Training and Test Datasets

The same training and test problems as for the Satallax evaluation were used. The strategy search took 2 weeks. 89 strategies were selected. LEO-II and LEO-MaLeS were run with a 300 second time limit per problem.

Of the 573 training problems 472 can be solved by LEO-II if the correct strategy is picked. LEO-MaLeS runs 5 start strategies, each with a 1 second time limit. Using more start strategies only marginally increases the number of problems solved by the start strategies. LEO-II's default mode solves 415 of the training problems (72.4 %), and 367 of the test problems (36.7 %). LEO-MaLeS improves this to 441 (77.0 %) and 417 (41.7 %) solved problems respectively. Figures 9 and 10 show the corresponding graphs. Figure 11 shows the results for those 699 problems that are not contained in the training problems.

[5]Description from the LEO-II website www.leoprover.org.

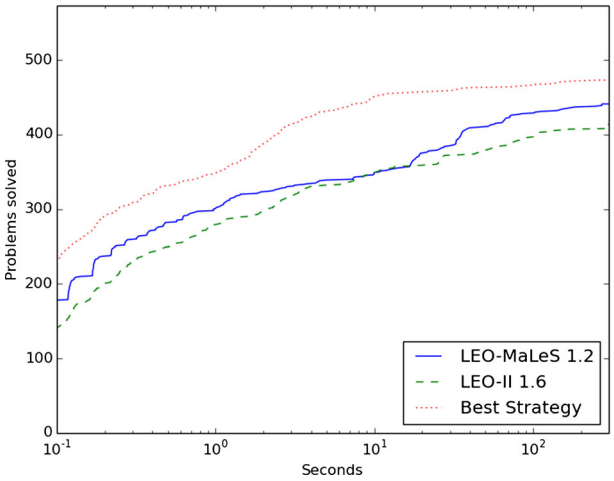

Fig. 9 Performance graph for LEO-MaLeS 1.2 on the training problems

Between 7 and 20 seconds, both provers solve approximately the same number of problems. For all other time limits, LEO-MaLeS solves more. On the test problems, a similar time frame turned out to be problematic for LEO-MaLeS. Between 5 and 30 seconds, LEO-II solves more problems than LEO-MaLeS. For other time limits, LEO-MaLeS solves more problems than LEO-II. This behavior indicates that the initial predictions of LEO-MaLeS are wrong. Better features could help remedy this problem. The sudden jump in the number of solved problems at around 30 seconds on the test dataset seems peculiar. Upon inspection, we found that 42 out of 43 problems solved in the $30 - 35$ seconds time frame are from the SEU (Set Theory) problem domain. These problems have very similar features and hence MaLeS creates similar strategy schedules. 34 of the 43 problems were solved by the same strategy.

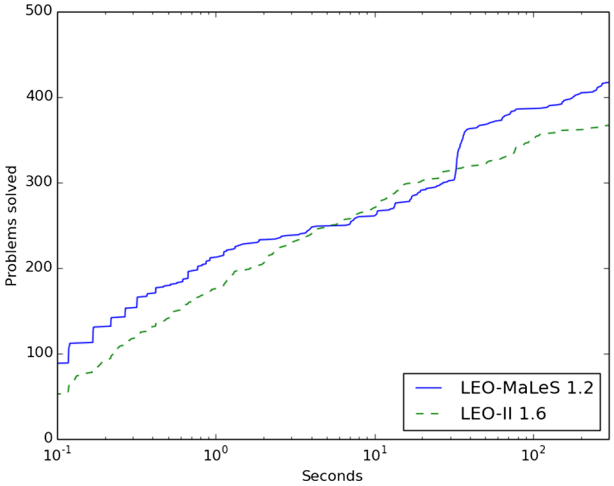

Fig. 10 Performance graph for LEO-MaLeS 1.2 on the test problems

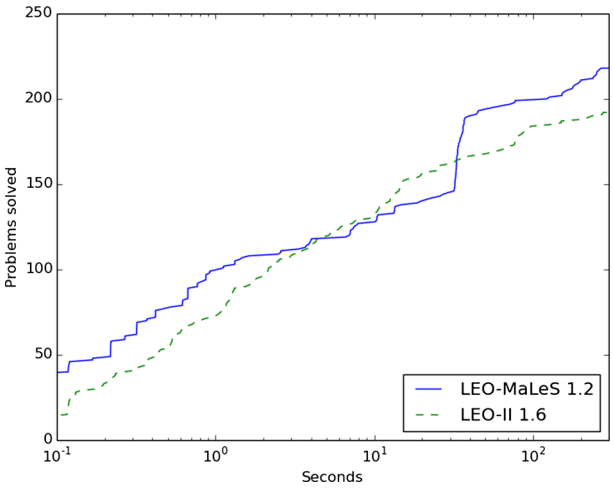

Fig. 11 Performance graph for Leo-MaLeS 1.2 on the unseen test problems

4.4 Further Remarks

There are a few issues to note that are independent of the underlying prover.

Multi-core Evaluations All the evaluations were done on multi-core machines, a 64 core AMD Opteron Processor 6276 with 1.4GHz per CPU and 256 GB of RAM and a 32 core Intel Xeon with 2.6GHz per CPU and 256 GB of RAM. All runtimes were measured in wall-clock time. During the evaluation we noticed irregularities in the runtime of the ATPs. When running a single instance of an ATP, the time needed to solve a problem often differed from the result we observed when running several instances in parallel, even when using less than the maximum number of cores. It turns out that the number of cores used during the evaluation heavily influences performance. The more cores, the worse the ATPs performed. We were not able to completely determine the cause of this behaviour, but the speed of the hard disk drive, shared cache and process swapping are all possible explanations. Reducing the hard disk drive load by changing the behavior of MaLeS from loading all models at the very beginning to only when they are needed did lead to more (and faster) solved problems. Eventually, all evaluation experiments (apart from the strategy searches for the sets of preselected strategies) were redone using only 20 out of 64 / 14 out of 32 cores and the results reported here are based on those runs.

How Good are the Predictions? Apart from the total number of solved problems, the quality of the predictions is also of interest. The training data of MaLeS is heavily biased because unsolvable problems are ignored (Section 3.3.1). Reducing the number of training problems during the update phase makes the predictions even less reliable. For some strategies, the average difference between the actual and predicted runtimes exceeds 40 seconds. Two heuristics were added to help MaLeS to deal with this uncertainty. First, the predicted runtime must always exceed the minimal runtime of the training data. This prevents unreasonably low (in particular negative) predictions. Second, if the number of training problems is less than a predefined minimum (set to 5) then the predicted runtime is the maximum runtime of the training data.

The Impact of the Learning Parameters Table 2 shows the learning parameters of MaLeS. *Tolerance*, *StartStrategies* and *StartStrategiesTime* had the greatest impact in our experiments. Tolerance influences the number of strategies used in MaLeS. A low value means more strategies, a high value less. For E and LEO, higher values ($1.0 - 15.0$ seconds) gave better results since fewer irrelevant strategies were run. Satallax performed slightly better with a low tolerance which is probably due to the fact that it can solve almost every problem in less than a second. The values for StartStrategies and StartStrategiesTime determine how many problems are left for learning. Ten StartStrategies with a 1 second $StartStrategiesTime$ are good default values for the provers tested. For LEO-II we found that the number of solved problems barely increased after 5 seconds, and hence changed the number of StartStrategies to 5.

5 Future Work

Apart from simplifying the installation and set up, there are several other ways to improve MaLeS. We present the most promising ones.

Automated Parameter Configuration Parameters like *Tolerance*, *StartStrategies* and *StartStrategiesTime* could and should be set automatically. We hope to implement this in the next version of MaLeS.

Features The quality of the runtime prediction function is limited by the quality of the features. Adding new features and/or integrating feature selection algorithms could increase the prediction capabilities of MaLeS.

Strategy Finding As an alternative to randomized hill climbing, different search algorithms should be supported. In particular simulated annealing and genetic algorithms seem promising. The biggest problem of the current implementation, the time it requires to find good strategies, could be improved by using a clusterized local search principle similar to the one employed in BliStr [26].

Strategy Prediction The runtime prediction functions are the heart of MaLeS. Machine learning offers dozens of different regression methods which could be used instead of the kernel methods of MaLeS. A big drawback of the current approach is that it scales badly due to the need to invert a new matrix after every tried strategy. One possible solution for eliminating the need for matrix computations and also the dependency on Numpy and Scipy would be a nearest neighbor algorithm.

6 Conclusion

Finding the best parameter settings and strategy schedules for an ATP is a time consuming task that often requires in-depth knowledge of how the ATP works. MaLeS is an automatic tuning framework for ATPs that, given the possible parameter settings of an ATP and a set of problems, finds good search strategies and creates individual strategy schedules. MaLeS currently supports E, LEO-II and Satallax and can easily be extended to work with other provers.

Experiments with the ATPs E, LEO-II and Satallax showed that the MaLeS version performs at least comparably to the respective default strategy selection algorithm. In some cases, the MaLeS optimized version solves considerably more problems than the untuned ATP.

MaLeS aims to simplify the workflow for both ATP users and developers. It allows ATP users to fine-tune ATPs to their specific problems and helps ATP developers to focus on actual improvements instead of time-consuming parameter tuning.

Acknowledgments The authors were supported by the Nederlandse organisatie voor Wetenschappelijk Onderzoek (NWO) projects "MathWiki: A Web-based Collaborative Authoring Environment for Formal Proofs" and "Learning2Reason". Christoph Benzmüller, Chad Brown, Stephan Schulz and Geoff Sutcliffe made this work possible by publicly releasing their programs and providing support whenever problems occurred. We would like to thank the anonymous reviewers, Jasmin Christian Blanchette, Michael Nahas and Pawel Sickinger for their helpful comments.

Appendix

Using MaLeS

MaLeS aims to be a general ATP tuning framework. In this section, we show how to setup E-MaLeS, LEO-MaLeS and Satallax-MaLeS, tuning any of those provers on new problems, and how to use MaLeS with a completely new prover. The first step is to clone the MaLeS git repository via

```
git clone https://code.google.com/p/males/
```

MaLeS requires Python 2.7, Numpy 1.6 or later, and Scipy 0.10 or later [12]. Installation instructions for Numpy and Scipy can be found at http://www.scipy.org/install.html.

E-MaLeS, LEO-MaLeS and Satallax-MaLeS

Setting up any of the presented systems can be done in three steps.

1. Install the ATP (E, LEO-II or Satallax)
2. Run the configuration script with the location of the prover as argument. For example

   ```
   EConfig.py --location=../E/PROVER
   ```

   for E-MaLeS.
3. Learn the prediction function via

   ```
   MaLeS/learn.py
   ```

After the installation, MaLeS can be used by running

```
MaLeS/males.py -t 30 -p test/PUZ001+1.p
```

where $-t$ denotes the time limit and $-p$ the problem to be solved.

Tuning E, LEO-II or Satallax for a New Set of Problems

Tuning an ATP for a particular set of problems involves finding good search strategies and learning prediction models. The search behavior is defined in the the file *setup.ini* in the main directory. Using the default search behavior, E, LEO-II and Satallax can be tuned for new problems as follows:

1. Install the ATP (E, LEO-II or Satallax)
2. Run the configuration script with the location of the prover as argument. For example

   ```
   EConfig.py --location=../E/PROVER
   ```

   for E-MaLeS.
3. Store the absolute pathnames of the problems in a new file with one problem per line and change the *PROBLEM* parameter in *setup.ini* to the file containing the problem paths.
4. Find promising strategies by searching with a short time limit (which is the default setup)

   ```
   MaLeS/findStrategies.py
   ```

5. (Optional) Run all promising strategies for a longer time. For this several parameters need to be changed.
   (a) Copy the value of *ResultsDir* to *TmpResultsDir*.
   (b) Copy the value of *ResultsPickle* to *TmpResultsPickle*.
   (c) Change the value of *ResultsDir* to a new directory.
   (d) Change the value of *ResultsPickle* to a new file.
   (e) Change *Time* in search to the maximal runtime (in seconds), e.g. 300.
   (f) Set *FullTime* to True.
   (g) Set *TryWithNewDefaultTime* to True.
6. (Optional) Run *findStrategies* again.

   ```
   MaLeS/findStrategies.py
   ```
7. The newly found strategies are stored in *ResultsDir*. MaLeS can now learn from these strategies via

   ```
   MaLeS/learn.py
   ```

For completeness, Table 2 contains a list of all parameters in *setup.ini* with their descriptions.

Using a New Prover

The behavior of MaLeS is defined in three configuration files: *ATP.ini* defines the ATP and its parameters, *setup.ini* configures the searching and learning of MaLeS and *strategies.ini* contains the default strategies of the ATP that form the starting point of the strategy search for the set of preselected strategies. To use a new prover, *ATP.ini* and *strategies.ini* need to be adapted. Table 3 describes the parameters in *ATP.ini*.

The section *Boolean Parameters* contains all flags that are given without a value. *List Parameters* contains flags which require a value and their possible values. MaLeS searches strategies in the parameter space defined by *Boolean Parameters* and *List Parameters*. Running *EConfig.py* creates the configuration file for E which can serve an example.

Table 2 Parameters of MaLeS

	Description
Settings Parameter	
`TPTP`	The TPTP directory. Not required.
`TmpDir`	Directory for temporary files.
`Cores`	How many cores to use.
`ResultsDir`	Directory where the results of the findStrategies are stored.
`ResultsPickle`	Directory where the models are stored.
`TmpResultsDir`	Like *ResultsDir*, but only used if *TryWithNewDefaultTime* is True.
`TmpResultsPickle`	Like *ResultsPickle*, but only used if *TryWithNewDefaultTime* is True.
`Clear`	If True, all existing results are ignored and MaLeS starts from scratch.
`LogToFile`	If True, a log file is created.
`LogFile`	Name of the log file.
Search Parameter	
`Time`	Maximal runtime during search.
`Problems`	File with the absolute pathnames of the problems.
`FullTime`	If True, the ATP is run for the value of *Time*. If False, it is run for the rounded minimal time required to solve the problem.
`TryWithNewDefaultTime`	If True, findStrategies uses the best strategies from *TmpResultsDir* and *TmpResultsPickle* as a start strategies for a new search.
`Walks`	How many different strategies are tried in the local search step.
`WalkLength`	Up to this many parameters are changed for each strategy in the local search step.
Learn Parameter	
`Features`	Which features to use. Possible values are *E* for the E features and *TPTP* for the TPTP features.
`FeaturesFile`	Location of the feature file.
`StrategiesFile`	Location of the strategies file.
`KernelFile`	Location of the file containing the kernel matrices.
`RegularizationGrid`	Possible values for λ.
`KernelGrid`	Possible values for σ.
`CrossValidate`	If False, no crossvalidation is done during learning. Instead the first values in *RegularizationGrid* and *KernelGrid* are used.
`CrossValidationFolds`	How many folds to use during crossvalidation.
`StartStrategies`	Number of start strategies.
`StartStrategiesTime`	Runtime of each start strategy.
`CPU Bias`	This value is added to each runtime before learning. Serves as a buffer against runtime irregularities.
`Tolerance`	For a strategy s to be considered as a good strategy, there must be at least one problem where the difference of the best runtime of any strategy and the runtime of s is at most this value.

Table 2 (continued)

	Description
Run Parameter	
`CPUSpeedRatio`	Predicted runtimes are multiplied with this value. Useful if the training was done on a different machine.
`MinRunTime`	Minimal time a strategy is run.
`Features`	Either *TPTP* for higher order features or *E* for first order features.
`StrategiesFile`	Location of the strategies file.
`FeaturesFile`	Location of the feature file.
`OutputFile`	If not None, the output of MaLeS is stored in this file.

Different ATPs have (unfortunately) different input formats for search parameters. MaLeS currently supports three formats: *E*, *LEO* or *Satallax*. Each format corresponds to the format of the respective ATP. Table 4 lists the differences. New formats need to be hardcoded in the file *Strategy.py*.

Strategies defined in *strategies.ini* are used to initialize the strategy queue during the strategy searching for the set of preselected strategies. The default ini format is used. Each strategy is its own section with each parameter on a separate line. For example

```
[NewStrategy12884]
FILTER_START = 0
ENUM_IMP = 100
INITIAL_SUBTERMS_AS_INSTANTIATIONS = true
E_TIMEOUT = 1
POST_CONFRONT3_DELAY = 1000
FORALL_DELAY = 0
LEIBEQ_TO_PRIMEQ = true
```

At least one strategy must be defined. After the ini files are adapted, the new ATP can be tuned and run using the procedure defined in the last two sections.

CASC Results

MaLeS 1.2 is the third iteration of the MaLeS framework. E-MaLeS 1.0 competed at CASC-23, E-MaLeS 1.1 at CASC@Turing and CASC-J6, and E-MaLeS 1.2 at CASC-24. Satallax-MaLeS competed for the first time at CASC-24. We give an overview of the older versions, the CASC performance and the changes over the years.

Table 3 Parameters in ATP.ini

ATP Settings Parameter	Description
`binary`	Path to the ATP binary.
`time`	Argument used to denote the time limit.
`problem`	Argument used to denote the problem.
`strategy`	Defines how parameters are given to the ATP. Three styles are supported: *E*, *LEO* and *Satallax*.
`default`	Any default parameters that should always be used.

Table 4 ATP Formats

Format	Description
`E`	Parameters and their values are joined by = if the parameter starts with --. Else the parameter is directly joined with its value. For example `---ordering=3 -sine13`.
`LEO`	Parameters and their values are joined by a space. For example `---ordering 3`.
`Satallax`	The parameters are written in a new mode file *M*. The ATP is then called with ATP `-m M`.

Table 5 Results of the FOF division of CASC 23

ATP	Vampire 0.6	Vampire 1.8	E-MaLeS 1.0	EP 1.4 pre
Solved	269/300	263/300	233/300	232/300
Average CPU Time	12.95	13.62	18.85	22.55

Table 6 Results of the FOF division of CASC-J6

ATP	Vampire 2.6	E-MaLeS 1.1	EP 1.6 pre	Vampire 0.6
Solved	429/450	377/450	359/450	355/450
Average CPU Time	13.17	17.85	13.46	11.81

Table 7 Results of the FOF division of CASC@Turing

ATP	Vampire 2.6	E-MaLeS 1.1	EP 1.6 pre	Vampire 0.6
Solved	469/500	401/500	378/500	368/500
Average CPU Time	20.26	20.81	14.49	16.40

Table 8 Results of the FOF division of CASC 24

ATP	Vampire 2.6	Vampire 3.0	EP 1.8	E-MaLeS 1.2
Solved	281/300	274/300	249/300	237/300
Average CPU Time	12.24	10.91	29.02	14.52

Table 9 Results of the THF division of CASC 24

ATP	Satallax-MaLeS 1.2	Satallax	Isabelle 2013
Solved	119/150	116/150	108/150
Average CPU Time	10.42	11.39	54.65

CASC-23

E-MaLeS 1.0 [10] was the first MaLeS version to compete at CASC. Stephan Schulz provided us with a set of strategies and information about their performance on all TPTP problems. This data was used to train a kernel-based classification model for each strategy. Given the features of a problem p, the classification models predict whether or not a strategy can solve p. Altogether, three strategies were run. First E's auto mode for 60 seconds, then the strategy with the highest probability of solving the problem as predicted by a Gaussian kernel classifier for 120 seconds. Finally the strategy with the highest probability of solving the problem as predicted by a linear (dot-product) kernel classifier was run for the remainder of the available time. E-MaLeS 1.0 won third place in the FOF division. Table 5 shows the results.

CASC@Turing and CASC-J6

E-MaLeS 1.1 [8] changed the learning from classification to regression. Like E-MaLeS 1.0, E-MaLeS 1.1 learned from (an updated version of) Schulz's data. Instead of predicting which strategy to run, E-MaLeS 1.1 learned runtime prediction functions. The learning method is the same as the one presented in this chapter, without the updating of the prediction functions. E-MaLeS 1.1 first ran E's auto mode for 60 seconds. Afterwards, each strategy was run for its predicted runtime, starting with the strategy with the lowest predicted runtime. E-MaLeS 1.1 won second place in the FOF divisions of both CASC@Turing (Table 6) and CASC-J6 (Table 7). It also came fourth in the LTB division of CASC-J6.

CASC-24

E-MaLeS 1.2 and Satallax-MaLeS 1.2 competed at CASC 24, both based on the algorithms presented in this chapter. E-MaLeS 1.2 used Schulz's strategies as start strategies for *find_strategies*. It is the first E-MaLeS that was not based on the CASC version of E (E 1.7 in E-MaLeS 1.2 vs E 1.8). E-MaLeS 1.2 got fourth place in the FOF division, losing to two versions of Vampire, and E 1.8. Several significant changes were introduced in E 1.8, in particular new strategies and E's own strategy scheduling. Satallax-MaLeS won first place in the THF division before Satallax. The results can be seen in Tables 8 and 9.

References

1. Benzmüller, C., Paulson, L.C., Theiss, F., Fietzke, A.: LEO-II - A cooperative automatic theorem prover for classical higher-order logic (System Description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) Automated Reasoning, Lecture Notes in Computer Science, vol. 5195, pp. 162–170. Springer (2008). doi:10.1007/978-3-540-71070-7_14
2. Bishop, C.M.: Pattern Recognition and Machine Learning. Information Science and Statistics. Springer (2006)

3. Bridge, J.P.: Machine learning and automated theorem proving. University of Cambridge, Computer Laboratory, Technical Report (792) (2010)
4. Brown, C.E.: Satallax: An Automatic Higher-Order Prover. In: Gramlich, B., Miller, D., Sattler, U. (eds.) Automated Reasoning, Lecture Notes in Computer Science, vol. 7364, pp. 111–117. Springer (2012). doi:10.1007/978-3-642-31365-3_11
5. Fuchs, M.: Automatic selection of search-guiding heuristics for theorem proving. In: Proceedings of the 10th Florida AI Research Society Conference, pp. 1–5. Florida AI Research Society (1998)
6. Guyon, I., Elisseeff, A.: An introduction to variable and feature selection. J. Mach. Learn. Res. **3**, 1157–1182 (2003)
7. Hutter, F., Hoos, H.H., Leyton-Brown, K., Stützle, T.: ParamILS: an automatic algorithm configuration framework. J. Artif. Intell. Res. **36**, 267–306 (2009). doi:10.1613/jair.2861
8. Kühlwein, D., Schulz, S., Urban, J.: E-MaLeS 1.1. In: Bonacina, M.P. (ed.) Automated Deduction – CADE-24, Lecture Notes in Computer Science, vol. 7898, pp. 407–413. Springer (2013). doi:10.1007/978-3-642-38574-2_28
9. Kohavi, R.: A study of cross-validation and bootstrap for accuracy estimation and model selection. In: Proceedings of the 14th International Joint Conference on Artificial Intelligence, vol. 2, pp. 1137–1143. Morgan Kaufmann Publishers Inc. (1995)
10. Kühlwein, D., Schulz, S., Urban, J.: Experiments with strategy learning for E Prover. In: 2nd Joint International Workshop on Strategies in Rewriting, Proving and Programming (2012)
11. Liu, H., Motoda, H.: Feature Selection for Knowledge Discovery and Data Mining. Kluwer Academic Publishers (1998). doi:10.1007/978-1-4615-5689-3
12. Oliphant, T.E.: Python for scientific computing. Comput. Sci. Eng. **9**(3), 10–20 (2007). doi:10.1109/MCSE.2007.58
13. Riazanov, A., Voronkov, A.: The design and implementation of VAMPIRE. AI Commun. **15**(2-3), 91–110 (2002)
14. Rifkin, R., Yeo, G., Poggio, T.: Regularized least-squares classification. In: Suykens, J., Horvath, G., Basu, S., Micchelli, C., Vandewalle, J. (eds.) Advances in Learning Theory: Methods, Model and Applications, pp. 131–154. IOS Press, Amsterdam (2003)
15. Schulz, S.: E—A Brainiac Theorem Prover. J. AI Commun. **15**(2-3), 111–126 (2002)
16. Shawe-Taylor, J., Cristianini, N.: Kernel Methods for Pattern Analysis. Cambridge University Press (2004)
17. Sutcliffe, G.: The TPTP problem library and associated infrastructure. J. Autom. Reason. **43**(4), 337–362 (2009). doi:10.1007/s10817-009-9143-8
18. Sutcliffe, G.: The CADE-22 automated theorem proving system competition - CASC-22. AI Commun. **23**(1), 47–60 (2010)
19. Sutcliffe, G.: The 5th IJCAR automated theorem proving system competition - CASC-J5. AI Commun. **24**(1), 75–89 (2011)
20. Sutcliffe, G.: The CADE-23 automated theorem proving system competition - CASC-23. AI Commun. **25**(1), 49–63 (2012)
21. Sutcliffe, G.: The 6th IJCAR automated theorem proving system competition—CASC-J6. AI Commun. **26**(2), 211–223 (2013)
22. Sutcliffe, G., Benzmüller, C.: Automated reasoning in higher-order logic using the TPTP THF infrastructure. J. Formalized Reason. **3**(1), 1–27 (2010)
23. Sutcliffe, G., Suttner, C.: Evaluating general purpose automated theorem proving systems. Artif. Intell. **131**(1-2), 39–54 (2001). doi:10.1016/S0004-3702(01)00113-8
24. Sutcliffe, G., Suttner, C.: The state of CASC. AI Commun. **19**(1), 35–48 (2006)
25. Gandalf, T.T.: J. Autom. Reason. **18**, 199–204 (1997). doi:10.1023/A:1005887414560
26. Urban, J.: BliStr: The Blind Strategymaker. CoRR (2013). arXiv:1301.2683
27. Vapnik, V.N.: The Nature of Statistical Learning Theory. Springer, New York (1995)
28. Wolf, A.: Strategy selection for automated theorem proving. In: Giunchiglia, F. (ed.) Artificial Intelligence: Methodology, Systems, and Applications, Lecture Notes in Computer Science, vol. 1480, pp. 452–465. Springer (1998). doi:10.1007/BFb0057466
29. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: portfolio-based algorithm selection for SAT. J. Artif. Intell. Res. **32**, 565–606 (2008). doi:10.1613/jair.2490